INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 725

THE INPUT / OUTPUT COMPLEXITY OF SORTING AND RELATED PROBLEMS

Alok AGGARWAL
Jeffrey Scott VITTER

SEPTEMBRE 1987

The Input/Output Complexity of Sorting and Related Problems

La complexité en entrées/sorties du tri et de quelques problèmes voisins

Alok Aggarwal[1] and *Jeffrey Scott Vitter*[2,3,4]

Abstract. We provide tight upper and lower bounds, up to a constant factor, for the number of inputs and outputs (I/Os) between internal memory and secondary storage required for five sorting-related problems: sorting, the fast Fourier transform (FFT), permutation networks, permuting, and matrix transposition. The bounds hold both in the worst case and in the average case. Secondary storage is modeled as a magnetic disk capable of transfering P blocks each containing B records in a single time unit; the records in each block must be input from or output to B contiguous locations on the disk. We give two optimal algorithms for the problems, which are variants of merge sorting and distribution sorting. In particular we show for $P = 1$ that the standard merge sorting algorithm is an optimal external sorting method, up to a constant factor in the number of I/Os. Our sorting algorithms use the same number of I/Os as does the permutation phase of key sorting, except when the internal memory size is extremely small, thus affirming the popular adage that key sorting is not faster. We also give a simpler and more direct derivation of Hong and Kung's lower bound for the FFT for the special case $B = P = O(1)$.

Résumé. Nous donnons des bornes inférieures et supérieures précises, à un facteur constant près pour le nombre d'entrées/sorties entre mémoire interne et mémoire secondaire que necessitent cinq problèmes reliés au tri: le tri, lâ transformation de Fourier rapide, les réseaux de permutation, la permutation d'enregistrements, et la transposition de matrices. Ces bornes sont valables dans le cas moyen et dans le cas le pire. La mémoire secondaire est modéliseé comme un disque magnétique capable de transférer P blocs contenant chacun B enregistrements, en une unité de temps; les enregistrements de chaque bloc doivent être traités comme B emplacements contigus sur disque. Nous donnons deux algorithmes optimaux qui sont des variantes du tri fusion et du tri distribution. En particulier, on montre que pour $P = 1$, le tri fusion usuel est une méthode de tri optimale, à un facteur constant près vis-à-vis du nombre d'entrées/sorties. Nos algorithmes de tri utilisent le même nombre d'entrées/sorties que la phase de permutation d'un tri sur une clé (sauf lorsque la mémoire interne est très petite). Ceci confirme l'adage d'après lequel le tri sur une clé n'est pas plus rapide qu'un tri fusion. Nous obtenons également une preuve simple et direct d'un résultat de Hong et Kung donnant des bornes inférieures pour la FFT dans le case où $B = P = O(1)$.

1. Introduction

The question of how to sort efficiently has been motivation for many studies in the analysis of algorithms and computational complexity, and rightly so, because the problem has strong practical and theoretical merit. Recent studies confirm that sorting continues to account for roughly one-fourth of all computer cycles. Much of those resources are consumed by external sorts, in which the file is too large to fit in internal memory and must reside in secondary storage (typically on magnetic disks). It is well documented that the bottleneck in external sorting is the time for input/output (I/O) between internal memory and secondary storage.

Sorts of extremely large size are becoming more and more common. For example, banks each night typically sort the demand deposits (checks) of the current day into increasing order by account number. Then the accounting files can be updated in a single linear pass through the sorted file. In many cases, banks are required to complete this processing before opening for the next business day. It is pointed out in [Lindstrom and Vitter, 1985] that a typical sort from a few years ago might involve a file of two million records, totalling 800 megabytes; sorting time would be 1–2 hours. But in the near future, typical large file sizes are expected to contain ten million records, totalling 10,000 megabytes, and current sorting methods would take most of one day to do the sorting. (Banks would then have trouble completing this processing before the next business day!)

Two alternatives for coping with this problem naturally present themselves: One approach is to relax the problem requirements and to investigate alternate computer architectures such as parallel or distributed systems. The other approach, which we take in this paper, is to examine the fundamental limits in terms of the number of I/Os for external sorting and related problems in current computing environments. We assume that there is a single central processing unit, and we model secondary storage as a generalized random-access magnetic disk. (For completeness, we also consider the case in which the disk has some parallel capabilities.) The parameters

---

[1] IBM Watson Research Center, P. O. Box 218, Yorktown Heights, N. Y. 10598, USA. Research was also done while the author was at the Mathematical Sciences Research Institute in Berkeley.

[2] Support was provided in part by NSF research grant DCR–84–03613, by an NSF Presidential Young Investigator Award with matching funds from an IBM Faculty Development Award and an AT&T research grant, and by a Guggenheim Fellowship.

[3] I. N. R. I. A., Domaine de Voluceau, Rocquencourt, B. P. 105, 78153 Le Chesnay Cedex, France. Research was also done while the author was at the Mathematical Sciences Research Institute in Berkeley, Brown University in Providence, and Ecole Normale Supérieure in Paris.

[4] To whom correspondence should be addressed, at Brown University, Dept. of Computer Science, Box 1910, Providence, R. I. 02912, USA.

of interest are

N = # records to sort;
M = # records that can fit into internal memory;
B = # records that can be transfered in a single block;
P = # blocks that can be transfered concurrently;

where $1 \le B \le M < N$ and $1 \le P \le \lfloor M/B \rfloor$. We denote the N records by R_1, R_2, ..., R_N. The parameters N, M, and B are refered to as the *file size*, *memory size*, and *block size*, respectively. Typical parameters for the two sorting examples mentioned earlier are $N = 2 \times 10^6$, $M = 2000$, $B = 100$, and $N = 10^7$, $M = 3000$, $B = 50$.

Each block transfer is allowed to access any contiguous group of B records on the disk. Parallelism is inherent in the problem in two ways: Each block can transfer B records at once, which models the well-known fact that a conventional disk can transfer a block of data via an I/O roughly as fast as it can transfer a single bit. The second factor is that there can be P block transfers at the same time, which models multiple I/O channels and read/write heads and "gather read—scatter write" capabilities of the disk.

Pioneering work in this area was done by Floyd [1972], who demonstrated matching upper and lower bounds of $\Theta\big((N \log N)/B\big)$ I/Os for the problem of matrix transposition for the special case $P = O(1)$, $B = \Theta(M) = \Theta(N^c)$, where c is a constant $0 < c < 1$. Floyd's lower bound for transposition also applied to the problems of permuting and sorting (since they are more general problems), and the bound matched the number of I/Os used by merge sort. For these restricted values of M, B, and P, the bound showed that essentially $\Omega(\log N)$ passes are needed to sort the file (since each pass takes $O(N/B)$ I/Os), and that merge sorting and the permutation phase of key sorting both perform the optimum number of I/Os. However, for other values of B, M, and P, Floyd's upper and lower bounds did not match, thus leaving open the general question of the I/O complexity of sorting.

In this paper we present optimal bounds, up to a constant factor, for all values of M, B, and P, for the following five sorting-related problems: sorting, fast Fourier transform (FFT), permutation networks, permuting, and matrix transposition. Permutating records is a dominant component of key sorting. The five problems are similar, but the lower bounds require different bents, which illustrate precisely the relation of the five problems to one another. The upper bounds can be obtained by a variant of merge sort with P-block lookahead forecasting and by a distribution sorting algorithm that uses a median finding subroutine. In particular, we can conclude that the dominant part of sorting, in terms of the number of I/Os, is the rearranging of the records, not determining their order, except when M is extremely small with respect to N. Thus key sorting typically requires as many I/Os as does general sorting.

Our results answer the pebbling questions posed in [Savage and Vitter, 1987] concerning the optimum I/O time needed to perform the computation implied by the FFT digraph (also called the butterfly or shuffle-exchange or Omega network). For lagniappe, we also give a simple direct proof of the lower bound for FFT when $B = P = O(1)$, which was previously proved in [Hong and Kung, 1981] using a complicated pebbling argument.

In the next section we define the sorting, FFT, permutation network, permutation, and matrix transposition problems. The I/O complexities for these problems are given in Section 3. We derive the lower bounds in Section 4 and give the algorithms in Section 5. Section 6 is devoted to the ad hoc proof of Hong and Kung's result. Conclusions are given in Section 7.

2. Problem Definitions

Let us picture the internal memory and secondary storage disk together as *extended memory*, consisting of a large array containing at least $M+N$ locations, each location capable of storing a single record. We arbitrarily number the M locations in internal memory by $x[1]$, $x[2]$, ..., $x[M]$ and the locations on the disk by $x[M+1]$, $x[M+2]$, ... The five problems can be phrased as follows:

Sorting

Problem Instance: The internal memory is empty, and the N records reside at the beginning of the disk; that is, $x[i] = \mathbf{nil}$, for $1 \le i \le M$, and $x[M+i] = R_i$, for $1 \le i \le N$.

Goal: The internal memory is empty, and the N records reside at the beginning of the disk in sorted nondecreasing order; that is, $x[i] = \mathbf{nil}$, for $1 \le i \le M$, and the records $x[M+1]$, $x[M+2]$, ..., $x[M+N]$ are ordered in nondecreasing order by their key values.

Fast Fourier Transform (FFT)

Problem Instance: Let us assume that N is a power of 2. The internal memory is empty, and the N records reside at the beginning of the disk; that is, $x[i] = \mathbf{nil}$, for $1 \le i \le M$, and $x[M+i] = R_i$, for $1 \le i \le N$.

Goal: The memory configuration is exactly as in the input, except that each record has $\log N$ "tags."

The number of tags represents the level of a node in the FFT digraph. The digraph has $1 + \log N$ levels of N nodes each; the first level contains the N input nodes, and the last level contains the N output nodes. Each non-input node has indegree 2, and each non-output node has outdegree 2. The FFT digraph for $N = 16$ is pictured in Figure 1. The FFT digraph is also known as the butterfly or shuffle-exchange or Omega network. We shall denote the ith node ($0 \le i \le N-1$) on level ℓ ($0 \le \ell \le \log N$) in the FFT digraph by $n_{\ell,i}$.† The two inputs to node $n_{\ell,i}$ are $n_{\ell-1,i}$ and $n_{\ell-1,i\oplus 2^{\ell-1}}$, where $\oplus$ denotes the exclusive-or operation. (Note that nodes $n_{\ell,i}$ and $n_{\ell,i\oplus 2^{\ell-1}}$ each have the same pair of inputs). Record R_i receives its ℓth tag when records R_i and $R_{i\oplus 2^{\ell-1}}$ reside in internal memory at the same time and both have $\ell - 1$ tags each.

Permutation Network

The *Problem Instance* and *Goal* are phrased the same as for FFT.

---

† Unless explicitly specified, the base of the logarithm is 2.

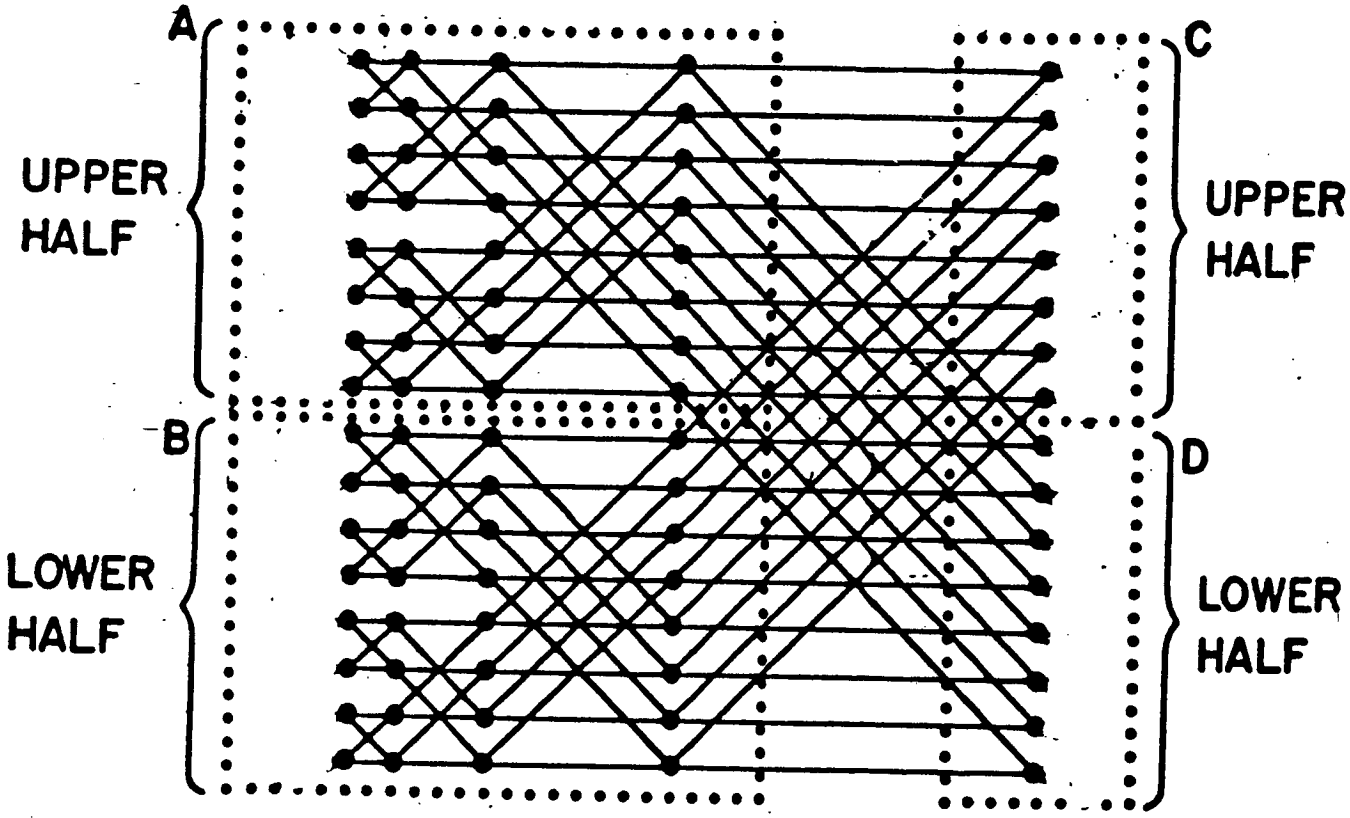

Figure 1. The FFT digraph for $N = 16$.

The permutation network digraph consists of N input nodes and some number $L \geq \log N$ of levels. The first level contains the N input nodes, and the last level contains the N output nodes. Each non-input node has indegree 2, and each non-output node has outdegree 2. For each of the $N!$ permutations $p_1, p_2, \ldots, p_N$, there is a set of N edge-disjoint paths such that, for $1 \leq i \leq N$, there is a path from the ith input node to the p_ith output node. Record R_i receives its ℓth tag when the records corresponding to its two input nodes reside in internal memory at the same time and each have $\ell - 1$ tags.

Permuting

The *Problem Instance* and *Goal* are the same as for Sorting, except that the key values of the N records are required to form a permutation of $\{1, 2, \ldots, N\}$.

There is a big difference between permutation networks and general permuting. In the latter case, the particular I/Os performed may depend upon the desired permutation, whereas with permutation networks all $N!$ permutations have to be generated by the same fixed sequence of I/Os.

Matrix Transposition

Problem Instance: A $p \times q$ matrix $A = (A_{i,j})$ of $N = pq$ records. The internal memory is empty, and the N records reside in row-major order at the beginning of the disk; that is, $x[i] = \text{nil}$, for $1 \leq i \leq M$, and $x[M + 1 + i] = A_{1+\lfloor i/q \rfloor, 1+i-q\lfloor i/q \rfloor}$, for $0 \leq i \leq N - 1$.

Goal: The internal memory is empty, and the transposed matrix A^{T} resides on disk in row-major order. (The transpose of A is the $q \times p$ matrix A^{T} defined by $A^{\mathrm{T}}_{i,j} = A_{j,i}$, for all $1 \leq i \leq q$ and $1 \leq j \leq p$.)

An equivalent goal is to store the original matrix A in column-major order on disk.

3. The Main Results

Our model requires that each block transfer in an input can move at most B records from disk into internal memory, and that the transfered records must come from a contiguous segment $x[M + i], x[M + i + 1], \ldots, x[M + i + B - 1]$ of B locations on the disk, for some $i > 0$; similarly, in each output the transfered records must be deposited within a contiguous segment of B locations. Our characterization of the I/O complexity for the five problems is given in the following main three theorems.

Theorem 3.1. *The average-case and worst-case number of I/Os required for sorting N records and for computing the N-input FFT graph is*

$$\Theta\left(\frac{N}{PB}\frac{\log(1+\frac{N}{B})}{\log(1+\frac{M}{B})}\right). \tag{3.1}$$

Similarly, the average-case and worst-case number of I/Os required for computing any N-input permutation network is

$$\Omega\left(\frac{N}{PB}\frac{\log(1+\frac{N}{B})}{\log(1+\frac{M}{B})}\right); \tag{3.2}$$

furthermore, there are permutation networks that can be computed with

$$O\left(\frac{N}{PB}\frac{\log(1+\frac{N}{B})}{\log(1+\frac{M}{B})}\right) \tag{3.3}$$

I/Os. We assume that records are indivisible; that is, bit manipulations like exclusive-oring are not allowed. For the sorting lower bound, the comparison model is assumed only for the case when M and B are extremely small, namely, when $B\log(1+M/B) = o(\log(1+N/B))$.

Theorem 3.2. *The average-case and worst-case number of I/Os required to permute N records, assuming the records are indivisible, is*

$$\Theta\left(\min\left\{\frac{N}{P}, \frac{N}{PB}\frac{\log(1+\frac{N}{B})}{\log(1+\frac{M}{B})}\right\}\right). \tag{3.4}$$

A good way to regard the expressions in the theorems is in terms of how many passes through the file are needed to solve the problem. A "linear-time" algorithm (defined to be one that requires a constant number of passes through the file) would use $O(N/PB)$ I/Os. The logarithmic factors that multiply the N/PB term in the above expressions indicate the degree of nonlinearity.

It is interesting to note that the optimum bound for sorting in Theorem 3.1 matches the second of the two terms being minimized in Theorem 3.2. When the second term in (3.4) achieves the minimum, which happens except when M and B are extremely small with respect to N, the problem of permuting is as hard as the more general problem of sorting; the dominant component of sorting in this case, in terms of the number of I/Os, is the routing of the records, not the determination of their order. When instead M and B are extremely small (namely, when $B\log(1+M/B) = o(\log(1+N/B))$), the N/P term in (3.4) achieves the minimum, and the optimum algorithm for permuting is to move the records in the naïve manner, one record per block transfer. This is precisely the case where advance knowledge of the output permutation makes the problem of permuting easier than sorting. The lower bound for sorting in Theorem 3.1 for this case requires the use of the comparison model.

Another interesting corollary comes from applying the bound for sorting in Theorem 3.1 to the case $M = 2$ and $B = P = 1$, where the number of I/Os corresponds to the number of comparisons needed to sort N records by a comparison-based internal sorting algorithm. Substituting $M = 2$ and $B = P = 1$ into (3.1) gives the well-known $\Theta(N\log N)$ bound.

Theorem 3.3. *The number of I/Os required to transpose a $p \times q$ matrix stored in row-major order, assuming the records are indivisible, is*

$$\Theta\left(\frac{N}{PB}\frac{\log\min\{M, 1+\min\{p,q\}, 1+\frac{N}{B}\}}{\log(1+\frac{M}{B})}\right). \tag{3.5}$$

When B is large, matrix transposition is as hard as general sorting, but for smaller B, the special structure of the transposition permutation makes transposing easier.

We should note that all our algorithms that achieve the upper bounds follow a more restrictive model of I/O, in which *only records that were output together in a single block can be input together in a single block.* This restriction allows our results for straightline programs (namely, for FFT and matrix transposition) to apply to the I/O model that is given in [Savage and Vitter, 1987], which is essentially the restricted model with $P = 1$.

4. Proof of the Lower Bounds

Without loss of generality, we can assume that B, M, and N are powers of 2 and that $B < M < N$. We shall consider the case $P = 1$ when there is only one I/O at a time; the general lower bound will follow by dividing the bound we obtain by P. It also suffices to consider the average case only, since the worst-case bound follows directly. For permuting and sorting, we assume that all $N!$ inputs are equally likely. The FFT, permutation network, and matrix transposition problems have no input distribution, so the average-case and worst-case models are the same.

Permuting

First we prove a useful lemma, which applies not only to permuting but also to the other problems. Let us call an input "simple" if each record transfered from disk is removed from the disk and deposited in an empty location in internal memory; similarly, an output is "simple" if the transfered records are removed from internal memory and deposited on empty locations on disk. The following lemma allows us to assume, for purposes of obtaining the lower bound, that all I/Os are simple; that is, there is exactly one copy of each record present throughout the computation.

Lemma 4.1. *For each computation that implements a permutation of the N records $R_1, R_2, \ldots, R_N$ (or that sorts or that transposes or that computes the FFT digraph or a permutation network), there is a corresponding computation strategy involving only simple I/Os such that the total number of I/Os is no greater.*

Proof. It is easy to construct the simple computation strategy by working backwards. We cancel the transfer of a record if its transfer is not needed for the final result. The resulting I/O strategy is simple. ∎

Our approach is to bound the number of possible permutations that can be generated by t I/Os. If we take the value of t for which the bound reaches $N!/2$, and then divide it by 2, we get a lower bound on the average number of I/Os. First let us define some terminology.

Definition 4.1. We say that a permutation $p_1, p_2, \ldots, p_N$ of the N records can be *generated at time t* if there is some sequence of t I/Os such that after the I/Os are performed the records appear in the correct permuted order in extended memory; that is, for all i, j, and k, we have

$$x[i] = R_{p_k} \text{ and } x[j] = R_{p_{k+1}} \quad \Longrightarrow \quad i < j.$$

The records do not have to be in contiguous positions in internal memory or on disk; there can be arbitrarily many empty locations between R_{p_k} and $R_{p_{k+1}}$.

Definition 4.2. We denote the kth set ($k \geq 1$) of B contiguous locations on the disk, namely, locations $x[M + (k-1)B + 1]$, $x[M + (k-1)B + 2]$, ..., $x[M + kB]$, as the *kth track*.

We make the following simplifying restrictions, which do not change the required number of I/Os by more than a small constant factor: As mentioned above, we assume that all I/Os are simple. The following preprocessing step is done before any other I/Os are performed: the N/B tracks are input into memory, in groups of one memoryload at a time; each memoryload of records is arbitrarily rearranged and output in its new order to consecutive positions on the disk. We require that each input and output transfer exactly B records, some of the records being possibly nil, and that the B records come from or go to a single track. For example, an input of $b < B$ records, with b_1 records from one track and $b_2 = b - b_1$ records from the next track, can be simulated using an internal memory of size $M + B$ by an input of the first track, an output of the $B - b_1$ records that are not needed (plus an additional b_1 nil records to take the place of the b_1 desired records), and then a corresponding input and output for the next track. As a consequence, since I/Os are simple, a track immediately after an input or immediately before an output must be empty. Internal computation time is not counted in our complexity model, so we can assume that the optimum algorithm, between I/Os, rearranges the records in internal memory however it sees appropriate.

After the preprocessing, the number of permutations generated is at most

$$(M!)^{N/M}. \tag{4.1}$$

Now let us consider the effect of an I/O. A tth output changes the number of permutations generated by at most a multiplicative factor of $N/B + t$, which can be bounded trivially by $N(\log N + 1)$. For the case of input, let us consider an input from a *specific track* on disk. Without loss of generality, we assume that the B records are deposited in locations $x[M - B + 1]$, ..., $x[M]$ in internal memory. These B records come from a single track on disk and were output together during some previous output. By our assumptions, this implies that at some previous time they were together in internal memory and were arranged in an arbitrary order by the algorithm. Thus, the $B!$ possible orders of the B inputed records could already have been generated before the input took place. This implies in a subtle way that the increase in the number of permutations generated due to rearrangement in internal memory is at most a multiplicative factor of $\binom{M}{B}$, which is the number of ways to intersperse B indistinguishable items within a group of size M.

The above analysis applies to input from a specific track. If the input is the tth I/O, there are at most $N/B + t - 1$ tracks to choose from for the I/O, plus one more because input from an empty track is also possible. Putting our results together, we find that the number of permutations generated at time t can be greater than the number of permutations generated at time $t - 1$ by at most a multiplicative factor of

$$\left(\frac{N}{B} + t\right)\binom{M}{B} \leq N(\log N + 1)\binom{M}{B}. \tag{4.2}$$

We get our lower bound by using (4.1) and (4.2) to determine the minimum value $T \geq$

N/B such that the number of permutations generated is at least $N!/2$:

$$(M!)^{N/M}\left(N(\log N+1)\binom{M}{B}\right)^{2T} \geq \frac{N!}{2}. \tag{4.3}$$

Taking logarithms and applying Stirling's formula to (4.3), with some algebraic manipulation, we get

$$\left(T-\frac{N}{B}\right)\left(\log N + B\log\frac{M}{B}\right) = \Omega\left(N\log\frac{N}{M}\right). \tag{4.4}$$

If $B\log(M/B) < \log N$, then it follows that $M < \sqrt{N}$ and from (4.4) we get

$$T = \Omega\left(\frac{N}{B} + \frac{N\log\frac{N}{M}}{\log N}\right) = \Omega(N). \tag{4.5}$$

On the other hand, if $\log N < B\log(M/B)$, then (4.4) gives us

$$T = \Omega\left(\frac{N}{B} + \frac{N\log\frac{N}{M}}{B\log\frac{M}{B}}\right) = \Omega\left(\frac{N\log\frac{N}{B}}{B\log\frac{M}{B}}\right). \tag{4.6}$$

Combining (4.5) and (4.6), we get

$$T = \Omega\left(\min\left\{N, \frac{N\log\frac{N}{B}}{B\log\frac{M}{B}}\right\}\right). \tag{4.7}$$

We get the lower bound in Theorem 3.2 by dividing (4.7) by P.

FFT and Permutation Networks

A key observation for obtaining the lower bound for the FFT is that we can construct a permutation network by stacking together three FFT digraphs, so that the output nodes of one FFT are the input nodes for the next [Wu and Feng, 1981]. Thus the FFT and permutation network problems are esentially equivalent, since as we shall see the lower bound for permutation networks matches the upper bound for FFT.

Let us consider an optimal I/O strategy for a permutation network. The second key observation is that the I/O sequence is *fixed*. This allows us to apply the lower bound proof developed above for permuting, with the helpful restriction that each I/O cannot depend upon the desired permutation; that is, regardless of the permutation, the records that are transfered during an I/O and the track accessed during the I/O are fixed for each I/O. Thus, each input changes the number of permutations generated by at most a multiplicative factor of $\binom{M}{B}$, rather than $(N/B+t)\binom{M}{B}$. Each output can at most double the number of permutations generated. The lower bound follows for $P = 1$ by finding the smallest number $T \geq N/B$ of I/Os such that

$$(M!)^{N/M}\binom{M}{B}^{T} \geq N!. \tag{4.8}$$

By using Stirling's formula, we get the same bound as in (4.6). Dividing by P gives the lower bound in Theorem 3.1.

It is interesting to note that since the I/O sequence is fixed and cannot depend upon the particular permutation, we are not permitted to use the naïve method of permuting, in which each block transfer moves one record from its initial to its final destination. This is reflected in the the growth rate of the number of permutations generated due to a single I/O: the $(N/B + t)$ term in the growth rate in (4.2) for permuting, which is dominant when the naïve method is optimal, does not appear in the corresponding growth rate for permutation networks.

Sorting

Permuting is a special case of sorting, so the lower bound for permuting in Theorem 3.2 also applies to sorting. However, when $B\log(M/B) = o\big(\log(N/B)\big)$, the lower bound becomes $\Omega(N/P)$, which is not good enough. In this case, the specific knowledge of what goes where makes generating a permutation easier than sorting.

We can get a better lower bound for sorting for the $B\log(M/B) = o\big(\log(N/B)\big)$ case by using an adversary argument, if we restrict ourselves to the comparison model of computation. Without loss of generality, let us make the following additional assumptions, similar to the ones earlier: All I/Os are simple. We preprocess the tracks one memoryload at a time, sort each memoryload, and then deposit the records in sorted order in consecutive locations on disk. Each I/O transfers B records, some possibly nil, to or from a single track on disk. We also assume that between I/Os the optimal algorithm performs all possible comparisons among the records in internal memory.

After the preprocessing, there are $N!/(M!)^{N/M}$ total orders consistent with the comparisons done so far. Let us consider an input of B records into internal memory. The B records come from a single track on disk, which previously was written in entirety via an output from internal memory. By our assumption, all comparisons were performed among the B records when they were together in internal memory. The records in internal memory before the input, which number at most $M - B$, have also had all possible comparisons peformed. Thus, after the input, there are at most $\binom{M}{B}$ possible outcomes to the comparisons between the records in memory. The adversary chooses the outcome that maximizes the number of total orders consistent with the comparisons done so far. It follows that the number of inputs needed in the worst case to sort is the minimum value of $T \geq N/B$ such that

$$\frac{N!}{(M!)^{N/M}\binom{M}{B}^T} \leq 1, \tag{4.9}$$

which yields the desired lower bound. Dividing by P gives the lower bound stated in Theorem 3.1.

The same result holds in the average-case model. We consider the comparison tree with $N!$ leaves, representing the $N!$ total orderings. Each node in the tree represents an input operation. After the preprocessing, the tree consists of $N!/(M!)^{N/M}$ subtrees, each having $(M!)^{N/M}$ leaves. In each of these subtrees, the nodes are constrained to have degree $\leq \binom{M}{B}$. The external path length divided by $N!$, minimized over all possible computation trees, gives the desired lower bound for $P = 1$. Dividing by P gives the lower bound of Theorem 3.1.

Matrix Tranposition

We prove the lower bound using a potential function argument similar to the one used by Floyd [1972]. It suffices to consider the case $P = 1$; the general lower bound will follow by dividing by P. Without loss of generality, we assume that p and q are powers of 2, and that all I/Os are simple and transfer exactly B records, some possibly nil.

We define the *ith target group* to be the set of records that will ultimately be in the ith track at the termination of the algorithm. We define the continuous function

$$f(x) = \begin{cases} x \log_e x, & \text{if } x > 0; \\ 0, & \text{if } x = 0. \end{cases} \tag{4.10}$$

We assign a charge

$$C_k(t) = \sum_{i,k \geq 1} f(x_{i,k}) \tag{4.11}$$

to the kth track at time t if after t I/Os the kth track contains $x_{i,k}$ records from the ith target group. Similarly, we assign a charge

$$C_{\mathrm{M}}(t) = \sum_{i \geq 1} f(y_i) \tag{4.12}$$

to the internal memory at time t, where y_i is the number of records from the ith target group that are in internal memory after the tth I/O. We define the potential at time t to be

$$POT(t) = C_{\mathrm{M}}(t) + \sum_{k \geq 1} C_k(t). \tag{4.13}$$

Let us denote by T the total number of I/Os that have been performed when the algorithm terminates. It is easy to verify that the values of $POT(t)$ before and after the algorithm are, respectively,

$$POT(0) = \begin{cases} 0, & \text{if } B < \min\{p, q\}; \\ N \log_e \dfrac{B}{\min\{p, q\}}, & \text{if } \min\{p, q\} \leq B \leq \max\{p, q\}; \\ N \log_e \dfrac{B^2}{N}, & \text{if } \max\{p, q\} < B; \end{cases} \tag{4.14}$$

$$POT(T) = N \log_e B. \tag{4.15}$$

If a block is output from internal memory to disk at time t, then the potential function does not increase at that point; that is, $POT(t) \leq POT(t-1)$. Let us assume that the tth I/O is an input from disk to internal memory. The increase in potential is bounded by

$$C_{\mathrm{M}}(t) - C_{\mathrm{M}}(t-1). \tag{4.16}$$

To bound (4.16), note that the number of records from a given target group that are in internal memory increases when some of the records were present in internal memory before the input and some others were included in the input. We use y_i' and y_i'' to

denote the number of records from the ith target group that are, respectively, present in internal memory at time $t-1$ and input into internal memory at time t. We have

$$C_M(t) - C_M(t-1) = \sum_{i \geq 1} (f(y_i' + y_i'') - f(y_i') - f(y_i'')), \tag{4.17}$$

where

$$\sum_{i \geq 1} y_i' \leq M - B \quad \text{and} \quad \sum_{i \geq 1} y_i'' \leq B. \tag{4.18}$$

A simple convexity argument shows that (4.17) is maximized when $y_i' = (M-B)B/N$ and $y_i'' = B^2/N$, for each $1 \leq i \leq N/B$. For $0 \leq y \leq x \leq 1$, we have

$$\begin{aligned} f(x+y) - f(x) - f(y) &= x \log_e \left(1 + \frac{y}{x}\right) + y \log_e \left(1 + \frac{x}{y}\right) \\ &\leq y + y \log_e \left(1 + \frac{x}{y}\right) \\ &= O\left(y \log \left(1 + \frac{x}{y}\right)\right). \end{aligned} \tag{4.19}$$

Substituting $x = (M-B)B/N$ and $y = B^2/N$ into (4.19), it follows from (4.17) that

$$C_M(t) - C_M(t-1) = O\left(B \log \frac{M}{B}\right). \tag{4.20}$$

At the end of the algorithm, we have $T \geq N/B$, and thus

$$T - \frac{N}{B} = \Omega\left(\frac{POT(T) - POT(0)}{B \log \frac{M}{B}}\right). \tag{4.21}$$

The lower bound

$$T = \Omega\left(\frac{N}{B} \frac{\log \min\{M, 1 + \min\{p, q\}, 1 + \frac{N}{B}\}}{\log(1 + \frac{M}{B})}\right) \tag{4.22}$$

follows by substituting (4.15) and the different cases of (4.14) into (4.21). The general lower bound in Theorem 3.3 for $P > 1$ follows by dividing (4.22) by P.

5. Optimal Algorithms

In this section we describe variants of merge sort and distribution sort that achieve the bounds in Theorems 3.1–3.3. As mentioned in Section 3, the algorithms follow the added restriction that records input in the same block must have been output previously in a single block, except for the first input of each track. It suffices to consider worst-case complexity, since the average-case result follows immediately. We first discuss the sorting problem and then apply our results to get optimum algorithms for permuting, FFT, permutation networks, and matrix transposition. Without loss of generality, we can assume that B, M, and N are powers of 2 and that $B < M < N$.

Merge Sorting

The standard merge sort algorithm works as follows: In the "run" formation phase, the N/B tracks are input into memory, in groups of one memoryload at a time; each memoryload is sorted into a "run," which is then output to consecutive positions on disk. At the end of the run formation phase, there are N/M runs on disk. In each pass of the merging phase, $M/B - 1$ runs are merged into one longer run. (Let us assume for simplicity that $M > 3B$; otherwise we reduce B appropriately, which increases the number of I/Os by at most a constant factor.) During the processing, one block from each of the runs being merged resides in internal memory. When the records of a block expire, the next track for that run is input. The resulting number of I/Os is

$$\frac{2N}{B}\left\lceil \log_{M/B-1}\frac{N}{M}\right\rceil = O\left(\frac{N\log\frac{N}{B}}{B\log\frac{M}{B}}\right). \tag{5.1}$$

This does not yield an optimal algorithm, however, when P is not bounded by a constant, since there is no way of knowing which P tracks should be input next. The solution is to modify the information that goes into each track. Besides the records themselves, we also place into each track $P - 1$ "endmarkers," which are the key values of the last record in the each of the next $P - 1$ tracks of the run. Using a generalization of the forecasting technique described in [Knuth, 1973], we can then determine the P tracks that will expire next. Note, however, that several of these tracks might not yet be present in internal memory. Merging proceeds until a track not currently in memory is needed. An input can then be performed to transfer the next P tracks needed, using the forecasting information, and the process continues.

First let us consider the case $P \le B/2$. In each pass, the endmarkers are not output at the same time that the track is output, since they are not yet determined at that time. Instead, when we output the records of the ℓth output track, we also output the endmarkers for the $(\ell - P)$th output track. To do that, we have to store in internal memory the addresses and the largest key values of the last $P - 1$ tracks. This consumes $O(B)$ space, under our assumption that $P \le B/2$, so the number of tracks and the number of I/Os needed to store a run of a given length do not change by more than a constant factor. The number of passes in the merging phase also does not change by more than a constant factor. The resulting speedup is $\Theta(P)$, as desired.

However, if $P > B/2$, then there may not be enough room to store the endmarkers without increasing the number of tracks per run by too large an amount. In

this case, we form "metatracks" of size $B' = \lceil \sqrt{2M} \rceil \geq B$. The number of metatracks that can be input concurrently is $P' = \lfloor PB/(\lceil B'/B \rceil B) \rfloor$, which is bounded by $M/B' \leq B'/2$. This satisfies the requirement for the construction in the previous paragraph, using P' and B' in place of P and B. The result is that the number of I/Os is reduced by $\Theta(P')$ from the number used by standard merge sort. By (5.1), the number of I/Os performed by the standard merge sort would be

$$O\left(\frac{N \log \frac{N}{B'}}{B' \log \frac{M}{B'}}\right). \tag{5.2}$$

Dividing (5.2) by $P' = PB/B'$ and with some algebraic manipulation, we get the desired upper bound stated in Theorem 3.1.

Distribution Sorting

For simplicity, we assume that M/B is a perfect square, and we use S to denote the quantity $\sqrt{M/B}$. The main idea in the algorithm is that with $O\big(N/(PB)\big)$ I/Os we can find S *approximate partitioning elements* $b_1, b_2, \ldots, b_S$ that break up the file into roughly equal-sized "buckets." (For completeness, we define the dummy partitioning elements $b_0 = -\infty$ and $b_{S+1} = +\infty$.) More precisely, we shall prove later, for $1 \leq i \leq S+1$, that the number of records whose key value is $\leq b_i$ is between $(i - \frac{1}{4})N/S$ and $(i + \frac{1}{4})N/S$. Hence, the number N_i of records in the ith bucket (that is, the number N_i of records whose key value K is in the range $b_{i-1} < K \leq b_i$) satisfies

$$\frac{1}{2}\frac{N}{S} \leq N_i \leq \frac{3}{2}\frac{N}{S}. \tag{5.3}$$

For the time being, let us assume that we can compute the approximate partitioning elements using $O\big(N/(PB)\big)$ I/Os. Then with $O\big(M/(PB)\big)$ additional I/Os we can input M records from disk into internal memory and partition them into the S bucket ranges. The records in each bucket range can be stored on disk in contiguous groups of B records each (except possibly for the last group) with a total of $O\big(M/(PB) + S/P\big) = O\big(M/(PB)\big)$ I/Os. This procedure is repeated for another $N/M - 1$ stages, in order to partition all N records into buckets. The ith bucket will thus consist of $G_i \leq N_i/B + N/M = O(N_i/B)$ groups of at most B contiguous records, by using inequality (5.3). The buckets are totally ordered with respect to one another. The remainder of the algorithm consists of recursively sorting the buckets one-by-one and appending the results to disk. The number of I/Os needed to input the contents of the ith bucket into internal memory during the recursive sorting is bounded by $G_i/P = O\big(N_i/(PB)\big)$. Let us define $T(n)$ to be the number of I/Os used to sort n records. The above construction gives us

$$T(N) = \sum_{1 \leq i \leq S+1} T(N_i) + O\left(\frac{N}{PB}\right). \tag{5.4}$$

Using the facts that $N_i = O(N/S) = O(N/\sqrt{M/B})$ and $T(M) = O\big(M/(PB)\big)$, we get the desired upper bound given in Theorem 3.1.

All that remains to show is how to get the S approximate partitioning elements via $O\big(N/(PB)\big)$ I/Os. Our procedure for computing the approximate partitioning

elements must work for the recursive step of the algorithm, so we assume that the N records are stored in $O(N/B)$ groups of contiguous records, each of size at most B. First let us describe a subroutine that uses $O(n/(PB))$ I/Os to find the record with the kth smallest key (or simply the kth smallest record) in a set containing n records, in which the records are stored on disk in at most $O(n/B)$ groups, each group consisting of $\leq B$ contiguous records. We load the n records into memory, one memoryload at a time, and sort each of the $\lceil n/M \rceil$ memoryloads internally. We pick the median record from each of these sorted sets and find the median of the medians using the linear-time sequential algorithm developed in [Blum, Floyd, Pratt, Rivest, and Tarjan, 1973]. The number of I/Os required for these operations is $O(n/(PB)+(n/B)/P+n/M) = O(n/(PB))$. We use the key value of this median record to partition the n records into two sets. It is easy to verify that each set can be partitioned into groups of size B (except possibly for the last list) in which each group is stored contiguously on disk. It is also easy to see that each of the two sets has size bounded by $3n/4$. The algorithm is recursively applied to the appropriate half to find the kth largest record; the total number of I/Os is $O(n/(PB))$.

We now describe how to apply this subroutine to find the S approximate partitioning elements in a set containing N records. As above, we start out by sorting N/M memoryloads of records, which can be done with $O(N/(PB) + (N/B)/P) = O(N/(PB))$ I/Os. Let us call the jth sorted set U_j. We construct a new set U' of size at most $4N/S$ consisting of the $\frac{1}{4}kS$th records (in sorted order) of U_j, for $1 \leq k \leq 4M/S - 1$ and $1 \leq j \leq N/M$. Each memoryload of M records contributes $4M/S > B$ records to U', so these records can be output one block at a time. The total number of contiguous groups of records comprising U' is $O(|U'|/B)$, so we can apply the subroutine above to find the record of rank $4iN/S^2$ in U' with only $O(|U'|/(PB)) = O(N/(SPB))$ I/Os; we call its key value b_i. The S b_i's can thus be found with a total of $O(N/(PB))$ I/Os. It is easy to show that the b_i's satisfy the conditions for being approximate partitioning elements, thus completing the proof.

Permuting

We use either of the sorting algorithms described above, unless $B\log(M/B) = o(\log(N/B))$, in which case it is faster to move the records one-by-one in the naïve manner to their final positions, using $O(N/P)$ I/Os.

FFT and Permutation Networks

As mentioned in Section 4, three FFT graphs concatenated together form a permutation network. So it suffices to consider optimum algorithms for FFT.

For simplicity, let us assume that $\log M$ divides $\log N$. The FFT digraph can be decomposed into $(\log N)/\log M$ stages, as pictured in Figure 2. Stage k, for $1 \leq k \leq (\log N)/\log M$, corresponds to the processing of levels $(k-1)\log M + 1$, $(k-1)\log M + 2, \ldots, k\log M$ in the FFT digraph, or equivalently to the acquisition of bits $(k-1)\log M + 1, (k-1)\log M + 2, \ldots, k\log M$ for each record. The M nodes on level $(k-1)\log M$ that share the same ancestors on level $k\log M$ are processed together in a phase. The corresponding M records are brought into internal memory

via a transposition permutation, and then the next $\log M$ tags for each record can be obtained.

The I/O requirement for each stage is thus due to the transpositions needed to rearrange the records into the proper groups of size M. The transpositions can be collectively done via a simple merging procedure described in the next subsection, which requires a total of $O\big((N/PB)\log_{M/B}\min\{M, N/B\}\big)$ I/Os. There are $(\log N)/\log M$ stages, making the total number of I/Os

$$O\left(\frac{N\log N}{PB\log M}\left(\frac{\log\min\left\{M,\frac{N}{B}\right\}}{\log\frac{M}{B}}\right)\right), \tag{5.5}$$

which can be shown by some algebraic manipulation to equal the upper bound of Theorem 3.1.

Matrix Transposition

Without loss of generality, we assume that p and q are powers of 2. Matrix transposition is a special case of permuting. The intuition gained from the lower bound proof in Section 4 can be used to develop a simple algorithm for achieving the upper bound in Theorem 3.3. In each track, the B records are partioned into different target groups; each group in the decomposition is called a *target subgroup*. Before the start of the algorithm, the size of each target subgroup is

$$x = \begin{cases} 1, & \text{if } B < \min\{p,q\}; \\ \dfrac{B}{\min\{p,q\}}, & \text{if } \min\{p,q\} \le B \le \max\{p,q\}; \\ \dfrac{B^2}{N}, & \text{if } \max\{p,q\} < B. \end{cases} \tag{5.6}$$

The algorithm uses a merging procedure. The records in the same target subgroup remain together throughout the course of the algorithm. In each pass, target subgroups are merged and become bigger. The algorithm terminates when each target subgroup is complete, that is, when each target subgroup has size B. In each pass, which takes $2N/PB$ I/Os, the size of each target subgroup increases by a multiplicative factor of $\Theta(M/B)$. The number of I/Os done by algorithm is thus

$$O\left(\frac{N}{PB}\log_{M/B}\frac{B}{x}\right). \tag{5.7}$$

We get the upper bound in Theorem 3.3 by substituting the values of x from (5.6) into (5.7).

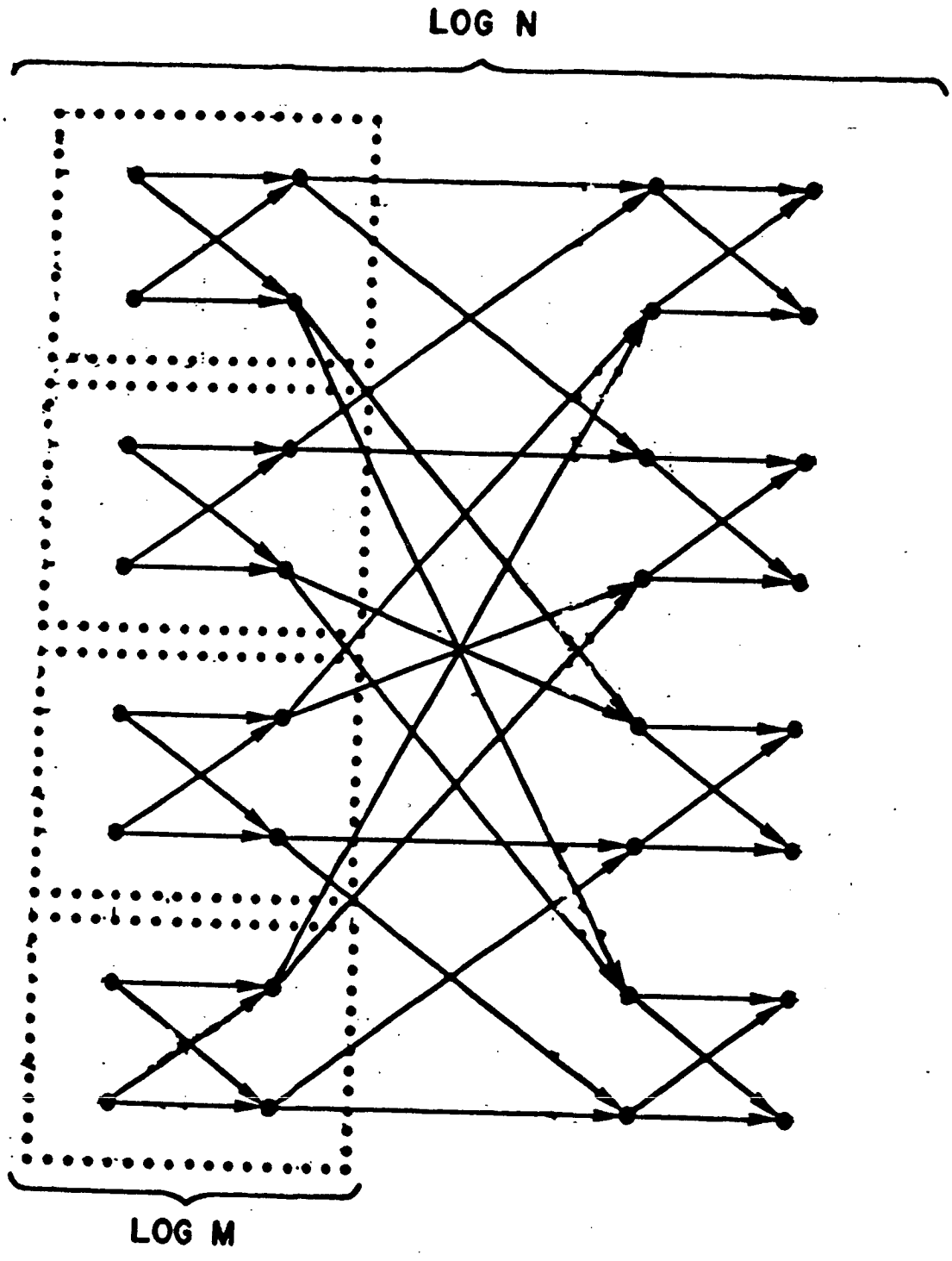

Figure 2. Decomposition of the FFT digraph into stages, for $N = 8$, $M = 2$.

6. Alternate Proof of Hong and Kung's Result

In this section we give a simple proof that the FFT requires $\Omega(N(\log N)/\log M)$ I/Os for the special case $B = P = O(1)$, which was proved in [Hong and Kung, 1981] using a complicated pebbling argument.

Our model for this special case can be phrased in terms of the red-blue pebble game, introduced in [Hong and Kung, 1981]. There are M red pebbles, representing internal memory storage, and an unlimited supply of blue pebbles, which represent information stored on disk. The FFT graph must be pebbled using the standard pebbling rules applied to the red pebbles, except that the following special I/O operations are allowed: A blue pebble may be placed on any node containing a red pebble, and a red pebble may be placed on any node containing a blue pebble, each at the cost of one I/O. The "cost" of the red-blue pebbling game is the number of I/Os performed; the red pebbling moves are free.

Our simplified proof of Hong and Kung's result rests on the following intuitive lemma:

Lemma 6.1. *Given any initial configuration of M red pebbles on the FFT digraph, at most $2M \log M$ red pebbling moves can be made without I/O.*

Proof. To bound the number of red pebbling moves, we use a dynamic charging strategy to allocate the moves to individual red pebbles. Let $num(p)$ denote the number of moves currently allocated to pebble p. A generic red pebbling move in the FFT digraph has the following form: Two pebbles p_1 and p_2 rest on nodes ℓ_1 and ℓ_2, and they share common parents u_1 and u_2. Both p_1 and p_2 are then moved to the upper level nodes u_1 and u_2. (Keeping one of the pebbles behind might only reduce the number of possible red pebbling moves, which we are trying to maximize.) Our charging strategy is to charge 1 to p_1 if $num(p_1) \leq num(p_2)$, and 1 to p_2 if $num(p_2) < num(p_1)$. The total number of red pebbling moves is therefore bounded by

$$2 \sum_{\text{pebbles } p} num(p). \tag{6.1}$$

The lemma below can be proved easily by induction; the proof is therefore omitted.

Lemma 6.2. *For each pebble p on node n in the FFT digraph, the number of nodes that contained a red pebble in the initial configuration and that are connected by a directed path to n is at least $2^{num(p)}$.*

There are M red pebbles, so each pebble p can "cover" at most M original placements. By Lemma 6.2, we have $num(p) \leq \log M$. Plugging this into (6.1) completes the proof of Lemma 6.1. ∎

Each node in the FFT digraph must be red pebbled at least once. Since there are $N \log N$ nodes, Lemma 6.1 implies that the number of I/Os required for the $P = M$, $B = 1$ case is at least

$$\frac{N \log N}{2M \log M}. \tag{6.2}$$

For the case $P = B = 1$, which is what we want to consider, we appeal to the following lemma.

Lemma 6.3. *Any fixed I/O schedule can be simulated by consecutive groups of I/O operations, in which each group consists either of M inputs or M outputs, and the total number of I/Os does not increase by more than a constant factor.*

Proof. The lemma follows easily from the fact that the I/O schedule is *fixed*, and thus caching can be done in order to group the I/Os in the desired fashion. ∎

If we treat each group of inputs and each group of outputs as a single operation, we find ourselves in the case $P = M$; a lower bound on the number of groups is given by (6.2). In terms of the $P = 1$ model, each group represents M I/Os, and our lower bound follows by multiplying (6.2) by M.

7. Conclusions

We have derived matching upper and lower bounds, up to a constant factor, for the average-case and worst-case number of I/Os needed to perform sorting-related tasks, which include sorting, FFT, permutation networks, permuting, and matrix transposition. In addition, the algorithms that achieve the upper bounds obey a more restrictive I/O rule that limits what records can be input and output together. If certain other restrictions are made on I/O, such as requiring that each block transfer must be simple and correspond to exactly one complete track, then the bounds are asymptotically tight in many cases; that is, the multiplicative factor between the upper and lower bounds is asymptotically 1. Our results also apply if the disk has a special "gather read—scatter write" capability that allows each block to be split up arbitrarily on the disk among S groups of contiguous records. This situation corresponds to a disk without the special capability that has block size $B' = B/S$ and degree of parallelization $P' = PS$.

Recently, Beigel and Gill [1986] have independently considered the problem of determining how many applications of a black box capable of sorting k records are necessary to sort N records. Their problem corresponds to the sorting problem for the case $P = M = k$ and $B = 1$. They have shown that $\Theta\big((N \log N)/(k \log k)\big)$ I/Os are optimal in that case (cf. Theorem 3.1). In addition, they have derived bounds on the constant factors involved in their version of the problem.

The optimal upper bound for $B = 1$ when $M = N^{\Omega(1)}$, can be obtained via a recursive application of Columnsort [Leighton, 1985]; however, for smaller M the upper bound is greater than optimal by a factor of roughly $\log \log N$.

Kwan and Baer [1985] study an alternative disk model, in which $P = 1$ and the disk is decomposed into contiguous cylinders, each composed of several tracks. (The track size is a hardware parameter, and can be different from the logical block size used for data transfer, unlike our use of the term in Definition 4.2.) The tracks all revolve at a constant rate. There is one read/write head per track, and the set of heads can move in unison from cylinder to cylinder. Seek time in an I/O is proportional to the number of cylinders traversed by the heads, and rotational latency time is proportional to the radial distance between the head positions at the start of an I/O request and the head positions at the beginning of the actual data transfer. An algorithm for permuting records is given that takes advantage of

locality of reference on the disk; it achieves better running times than merge sort in this model when the file size is large.

However, we believe that the simpler model we use in this paper gives more meaningful results, because the model of [Kwan and Baer, 1985] is overly pessimistic in how it models a random seek, in comparison with current technology. For example, for the large-capacity magnetic disks made by IBM, the time to do a seek between adjacent cylinders is of the same order of magnitude as the time for a random seek or for a complete revolution. In this more realistic context, the permutation algorithm of [Kwan and Baer, 1985] is slower than merge sort. In addition, the I/O block size in large external sorts is often on the order of the disk track size, Thus, the time for the data transmission during an I/O is as large in magnitude as the seek and latency times, which justifies the simpler model we study in this paper.

We conclude this paper with a challenging open problem: It would be nice to remove from the lower bound proofs the assumption that the records must be indivisible and to allow, for example, arbitrary bit manipulations and dissections of the records. Intuitively, the lower bound should still hold in this more general model, since it is unlikely that these operations are of any great help, but no proof of the lower bound is known. Such a proof would no doubt provide great insight into the nature of information transfer and sorting-related computations.

References

R. Beigel and J. Gill. Personal communication (1986).

M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. "Time Bounds for Selection," *Journal of Computer and System Sciences*, **7** (1973), 448–461.

R. W. Floyd. "Permuting Information in Idealized Two-Level Storage," in *Complexity of Computer Calculations*, edited by R. Miller and J. Thatcher, Plenum, New York (1972), 105–109.

Hong J. W. and H. T. Kung. "I/O Complexity: The Red-Blue Pebble Game," *Proceedings of the 13th Annual ACM Symposium on Theory of Computing*, Milwaukee, WI (October 1981), 326–333.

D. E. Knuth. *The Art of Computer Programming*. Volume III: *Sorting and Searching*. Addison-Wesley, Reading, MA (1973).

S. C. Kwan and J. L. Baer. "The I/O Performance of Multiway Mergesort and Tag Sort," *IEEE Transactions on Computers*, C–**34**(4), Special Issue on Sorting (April 1985), 383–387.

F. T. Leighton. "Tight Bounds on the Complexity of Parallel Sorting," *IEEE Transactions on Computers*, C–**34**(4), Special Issue on Sorting (April 1985).

E. E. Lindstrom and J. S. Vitter. "The Design and Analysis of BucketSort for Bubble Memory Secondary Storage," *IEEE Transactions on Computers*, C–**34**(3) (March 1985), 218–233.

J. E. Savage and J. S. Vitter. "Parallelism in Space-Time Tradeoffs," *Advances in Computing Research*, Volume 4: Special Issue on Parallel and Distributed Computing, JAI Press, Greenwich, CT (1987), 117–146.

C. L. Wu and T. Y. Feng. "The Universality of the Shuffle-Exchange Network," *IEEE Transactions on Computers*, C-30(5) (May 1981), 324–332.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique